BOSSCODER ACADEMY

SPRING BOOT

Interview Questions

Crack Technical Interviews

Disclaimer

Everyone learns at their own pace.

What matters most is your dedication and consistency.

This guide is designed to help you practice common Spring Boot concepts which are commonly asked to help you excel in your technical interviews.

Q 1. Explain the role of Spring Boot in microservices architecture.

Ans.

Spring Boot simplifies the development and deployment of microservices by providing a suite of tools and frameworks that streamline the setup and configuration processes. It offers features like embedded servers (Tomcat, Jetty), production-ready metrics, health checks, and externalized configuration to facilitate easy management of multiple microservices. Spring Boot's opinionated approach reduces boilerplate code and integrates seamlessly with Spring Cloud to manage distributed systems concerns such as service discovery, circuit breakers, and load balancing.

Q 2. How does Spring Boot achieve auto-configuration and what are its limitations?

Ans.

Spring Boot achieves auto-configuration by using '@EnableAutoConfiguration' and a set of conditional annotations like '@ConditionalOnClass', '@ConditionalOnMissingBean', etc., to automatically configure Spring Beans based on the classpath settings, existing beans, and various property settings. The 'META-INF/spring.factories' file specifies the '@Configuration' classes to be auto-configured. Limitations include potential conflicts with manual configurations, difficulty in debugging configuration issues, and the possibility of unnecessary components being loaded if not properly excluded.

Q 3. Discuss the significance of Spring Boot's actuator module in production environments.

Ans.

The Spring Boot Actuator module provides essential tools for monitoring and managing Spring Boot applications in production. It exposes various endpoints (like '/actuator/health', '/actuator/info', '/actuator/metrics') that offer insights into application health, metrics, environment properties, and more. This helps administrators and developers to quickly diagnose issues, monitor performance, and ensure the application is running as expected. Actuator can be integrated with monitoring systems and is customizable to expose only the necessary endpoints.

Q 4. How does Spring Boot support security, and what are the best practices for securing a Spring Boot application?

Ans.

Spring Boot integrates with Spring Security to provide comprehensive security solutions, including authentication, authorization, and protection against common vulnerabilities like CSRF, XSS, and SQL Injection. Best practices for securing a Spring Boot application include:

- Using HTTPS for encrypted communication.
- Storing sensitive data securely using encryption and secure credential management.

- Implementing strong authentication mechanisms, such as OAuth2 and JWT.
- Regularly updating dependencies to patch known vulnerabilities.
- Employing role-based access control (RBAC) to restrict access based on user roles.
- Utilizing security headers (e.g., Content-Security-Policy, X-Content-Type-Options) to prevent attacks.
- Conducting regular security audits and code reviews.

Q 5. What are Spring Boot starters and how do they simplify dependency management?

Ans.

Spring Boot starters are a set of convenient dependency descriptors that simplify dependency management by aggregating commonly used dependencies into a single starter package. For example, 'spring-boot-starter-web' includes dependencies for Spring MVC, Jackson, and an embedded Tomcat server. This abstraction reduces the need to specify individual dependencies and ensures compatibility among included libraries. Starters streamline the setup process and provide a consistent base for application development, allowing developers to focus more on business logic than on configuration.

Q 6. How do you create a simple RESTful web service using Spring Boot?

Ans.

1. Create a Spring Boot application using Spring Initializr or your preferred method.
2. Add the ‘spring-boot-starter-web’ ‘dependency’ to your ‘pom.xml’ or ‘build.gradle’ file.
3. Create a controller class with ‘@RestController’ annotation.
4. Define a request mapping method using ‘@GetMapping’, ‘@PostMapping’, etc.

```java
@RestController
@RequestMapping("/api")
public class MyController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Q 7. How do you connect a Spring Boot application to a MySQL database?

Ans.

1. Add the 'mysql-connector-java' and 'spring-boot-starter-data-jpa' dependencies to your 'pom.xml'.
2. Configure the database connection in 'application.properties':

```properties
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
spring.jpa.hibernate.ddl-auto=update
```

3. Create an entity class and a repository interface:

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
```

```
    // getters and setters
}

public interface UserRepository extends
JpaRepository<User, Long> {

}
```

Q 8. How do you handle exceptions globally in a Spring Boot application?

Ans.

1. Use *@ControllerAdvice* and *@ExceptionHandler* to handle exceptions globally:

```java
@ControllerAdvice
public class GlobalExceptionHandler {

      @ExceptionHandler(ResourceNotFoundException.class)
      public ResponseEntity<String>
      handleResourceNotFound(ResourceNotFoundException
      ex) {
        return new ResponseEntity<>(ex.getMessage(),
        HttpStatus.NOT_FOUND);
      }
```

```
    @ExceptionHandler(Exception.class)
 public ResponseEntity<String>
      handleGeneralException(Exception ex) {
         return new ResponseEntity<>("An error
         occurred",
         HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Q 9. How do you use Spring Boot DevTools to enhance the development experience?

Ans.

1. Add '*spring-boot-devtools*' dependency to your '*pom.xml*' :

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

DevTools enables automatic restarts, live reload, and configurations for faster development. Ensure your IDE is configured to compile the code on save.

Q 10. How do you configure a custom endpoint in Spring Boot Actuator?

Ans.

Implement a custom actuator endpoint by creating a bean of type Endpoint

```java
@Component
public class CustomEndpoint {


    @ReadOperation
    public String customEndpoint() {
        return "Custom Endpoint Output";
    }
}
```

Enable the endpoint in *'application.properties'* :

```properties

management.endpoints.web.exposure.include=custom
```

Q 11. How do you implement pagination and sorting in a Spring Boot application?

Ans.

Use Spring Data JPA's PagingAndSortingRepository

```java
public interface UserRepository extends
PagingAndSortingRepository<User, Long> {
}
```

In your service or controller, use '*Pageable*' and '*PageRequest*' :

```java
@GetMapping("/users")
public Page<User> getUsers(Pageable pageable) {
    return userRepository.findAll(pageable);
}
```

Q 12. How do you implement pagination and sorting in a Spring Boot application?

Ans.

Use Spring Data JPA's PagingAndSortingRepository

```java
public interface UserRepository extends
PagingAndSortingRepository<User, Long> {
}
```

In your service or controller, use '*Pageable*' and '*PageRequest*' :

```java
@GetMapping("/users")
public Page<User> getUsers(Pageable pageable) {
    return userRepository.findAll(pageable);
}
```

Q 13. How do you configure and use a custom banner in a Spring Boot application?

Ans.

Place a banner.txt file in the src/main/resources directory with your custom banner text. You can use ASCII art generators to create the text.

Q 14. How do you integrate Spring Boot with Kafka?

Ans.

Add the '*spring-kafka*' dependency to your '*pom.xml*' :

```xml
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Configure Kafka properties in '*application.properties*' :

```xml
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=my-group
```

Create Kafka producer and consumer beans:

```java
@Bean
public KafkaTemplate<String, String> kafkaTemplate() {
    return new KafkaTemplate<>(producerFactory());
}

@KafkaListener(topics = "myTopic", groupId = "my-group")
public void listen(String message) {
    System.out.println("Received: " + message);
}
```

Q 15. How do you schedule tasks in Spring Boot?

Ans.

Enable scheduling with *@EnableScheduling* and use *@Scheduled* annotation:

```java
@SpringBootApplication
@EnableScheduling
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

@Component
public class ScheduledTasks {
    @Scheduled(fixedRate = 5000)
    public void performTask() {
       System.out.println("Scheduled task running...");
    }
}
```

Q 16. How do you configure a Spring Boot application to use Spring Security?

Ans.

Add spring-boot-starter-security dependency and configure security

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http)
    throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin().permitAll()
            .and()
            .logout().permitAll();
    }
}
```

Q 17. How do you implement caching in a Spring Boot application?

Ans.

Add '*spring-boot-starter-cache*' dependency and enable caching with *@EnableCaching*:

```java
@SpringBootApplication
@EnableCaching
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class,
args);
    }
}

@Service
public class UserService {
    @Cacheable("users")
    public User getUserById(Long id) {
      return userRepository.findById(id).orElse(null);
    }
}
```

Configure cache in '*application.properties*' :

```properties
spring.cache.type=simple
```

Q 18. How do you implement caching in a Spring Boot application?

Ans.

Add '*spring-boot-starter-cache*' dependency and enable caching with *@EnableCaching*:

```java
spring.mail.host=smtp.example.com
spring.mail.port=587
spring.mail.username=myusername
spring.mail.password=mypassword
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

Create a service to send emails:

```java
@Service
public class EmailService {
    @Autowired
    private JavaMailSender mailSender;

    public void sendSimpleEmail(String to, String
    subject, String text) {
      SimpleMailMessage message = new
      SimpleMailMessage();
      message.setTo(to);
      message.setSubject(subject);
      message.setText(text);
      mailSender.send(message);
    }
}
```

Q 19. How do you implement internationalization (i18n) in a Spring Boot application?

Ans.

Configure message source in '*application.properties*' :

```properties
spring.messages.basename=messages
spring.messages.encoding=UTF-8
```

Create message properties files (e.g., *messages_en.properties, messages_fr.properties*) in *src/main/resources*.
Use *'MessageSource'* to access messages:

```java
@RestController
public class GreetingController {
    @Autowired
    private MessageSource messageSource;

    @GetMapping("/greet")
    public String greet(Locale locale) {
        return messageSource.getMessage("greeting",
        null, locale);
    }
}
```

Q 20. How do you integrate a third-party library (like Lombok) in a Spring Boot application?

Ans.

Add the Lombok dependency to your '*pom.xml*' :

```xml
<dependency>
    <groupId>org.projectlombok</
groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.20</version>
    <scope>provided</scope>
</dependency>
```

Use Lombok annotations in your classes:

```java
@Data
@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

Q 21. How do you create a custom validator in Spring Boot?

Ans.

Create a custom annotation and validator

```java
@Target({ ElementType.FIELD, ElementType.METHOD,
ElementType.PARAMETER, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = CustomValidator.class)
@Documented
public @interface CustomConstraint {
    String message() default "Invalid value";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public class CustomValidator implements
ConstraintValidator<CustomConstraint, String> {
    @Override
    public void initialize(CustomConstraint
    constraintAnnotation) {
    }
```

```
    @Override
    public boolean isValid(String value,
    ConstraintValidatorContext context) {
        return value != null && value.matches("[A-Z]{2}
[0-9]{4}");
    }
}
```

Use the custom validator in your model:

```java
public class MyModel {
    @CustomConstraint
    private String customField;
}
```

BOSSCODER ACADEMY

WHY BOSSCODER?

- **2200+** Alumni placed at Top Product-based companies.
- More than **136% hike** for every **2 out of 3** working professional.
- Average package of **24LPA.**

> The syllabus is most **up-to-date** and the list of problems provided covers **all important topics.**

Lavanya
Meta

> Course is very well **structured** and **streamlined** to crack any **MAANG** company

Rahul
Google

EXPLORE MORE